Chapter 3 Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars

Copyright © 2018 Pearson. All rights reserved.

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition

Copyright © 2018 Pearson. All rights reserved.

Example

- Syntax

  while (boolean_expr) statement

- Sematic

  When the current value of the Boolean expression is true, the embedded statement is executed. Then control implicitly returns to the Boolean expression to repeat the process. If the Boolean expression is false, control transfers to the statement following the while construct.

Copyright © 2018 Pearson. All rights reserved.

Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

Copyright © 2018 Pearson. All rights reserved.

The General Problem of Describing Syntax: Terminology/Examples

Consider the following Java statement:

```
index = 2 * count + 17;
```

The lexemes and tokens of this statement are:

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Copyright © 2018 Pearson. All rights reserved.

Formal Definition of Languages

- **Recognizers**
- Focuses on analyzing existing code to determine its adherence to the syntax rules of a programming language.

- A recognizer checks whether a program is **syntactically valid** according to a grammar.

  - Example: A parser, that checks whether {+ x 2} follows your grammar:

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         | <var>
```

Copyright © 2018 Pearson. All rights reserved.

Formal Definition of Languages

Generators

- A device that generates valid sentences in a language

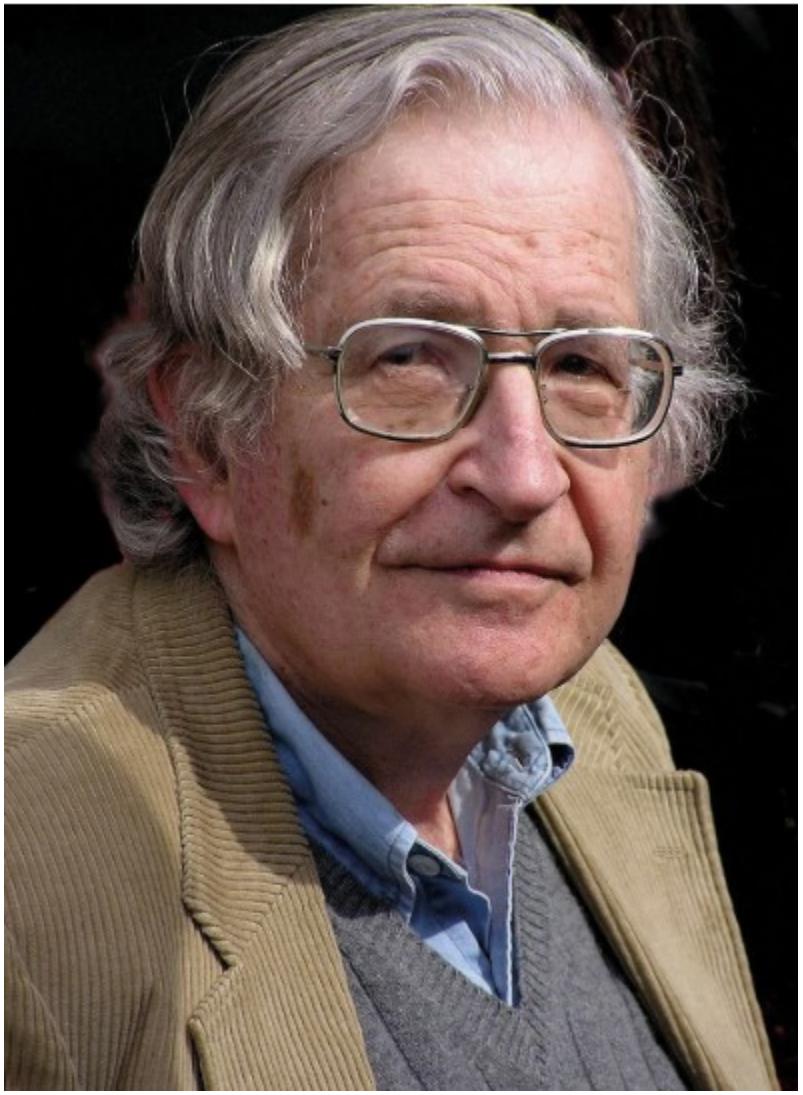
```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         | <var>
```

```
1
{+ 1 2}
{let1 {x 5} {+ x 1}}
```

Copyright © 2018 Pearson. All rights reserved.

Context-Free Grammars and BNF

- Context-Free Grammars
- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages



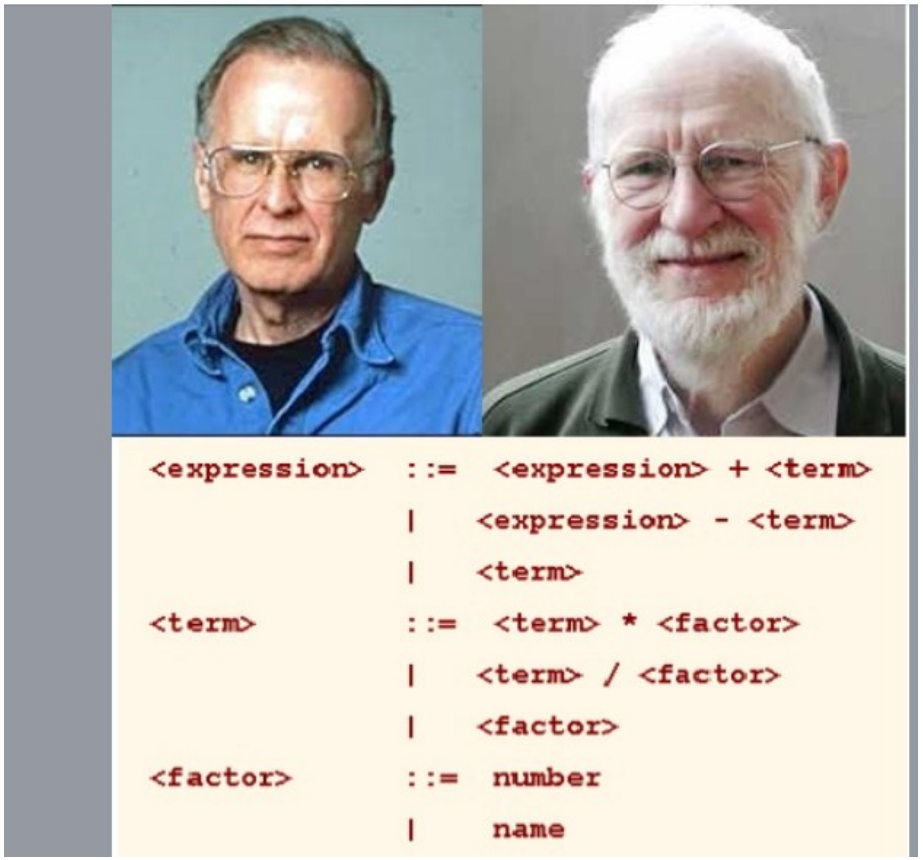
Copyright © 2018 Pearson. All rights reserved.

BNF (Backus-Naur Form)

Backus-Naur Form (1959)

- Invented by John Backus to describe the syntax of Algol 58
- BNF is equivalent to context-free grammars
- A set of production rules that describe how valid strings in the language can be formed

Backus Naur Form (BNF) is a famous notation technique for context-free grammars, and used to describe the syntax of programming languages, named after the scientists John Backus, 19 (left) and Peter Naur, 1928-2016 (right).



```
<expression>  ::=  <expression> + <term>
              |    <expression> - <term>
              |    <term>
<term>        ::=  <term> * <factor>
              |    <term> / <factor>
              |    <factor>
<factor>      ::=  number
              |    name
```

Copyright © 2018 Pearson. All rights reserved.

BNF Fundamentals

- Elements: syntactic variables or *nonterminal symbols (place holders), and* just *terminals (tokens)*
- Each rule has the following form LHS->RHS
- LHS (left-hand side) is an abstraction and is a nonterminal symbol
- A nonterminal symbol is replaced by one or more sequences of other terminal and/or nonterminal symbols
- RHS (right-hand side) represents a sequence of terminals (actual symbols of the language) and/or non-terminals

Copyright © 2018 Pearson. All rights reserved.

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier
             | identifier, <ident_list>
```

What language does it generate?

```
total                  → identifier
```

total, sum

total, sum, avg

A grammar is **context-free** if every production has **one nonterminal** on the left-hand side. Here, the LHS is always just <ident_list>, so it's context-free.

Copyright © 2018 Pearson. All rights reserved.

Example

- Examples of BNF rules:

```
<ident_list> → identifier
             | identifier, <ident_list>
```

total, sum, avg ?

→ identifier , <ident_list>
→ total , <ident_list>
→ total , identifier , <ident_list>
→ total , sum , <ident_list>
→ total , sum , identifier
→ total , sum , avg

Copyright © 2018 Pearson. All rights reserved.

BNF Describing a list

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier
             | identifier, <ident_list>

<ident_list> ::= identifier
              | identifier , <ident_list>
```

x, ?
,x ?

Copyright © 2018 Pearson. All rights reserved.

Grammers and derivations

- A grammar is a generative device for defining languages.
- The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol
- This sequence of rule applications is called a derivation

```
<ident_list> → identifier
             | identifier, <ident_list>
```

Copyright © 2018 Pearson. All rights reserved.

An Example Grammar for a small Language

```
<program> → <stmts>
<stmts> →  <stmt>
         |  <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var>  → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var>
        | const
```

a = b + const ?

Copyright © 2018 Pearson. All rights reserved.

An Example Grammar for a small Language

```
<program> → <stmts>
<stmts> →  <stmt>
         |  <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var>  → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var>
        | const
```

a = b + const ; c = d – a ? <stmts> → <stmt> ; <stmts>

<var> = <expr>; <stmts>

a = <expr>; <stmts>

a = <term> + <term>; <stmts>

a = <var> + <term>; <stmts>

a = b + <term>; <stmts>

a = b + const; <stmts>

….

Copyright © 2018 Pearson. All rights reserved.

An Example Grammar for a small Language

```
<program> → <stmts>
<stmts> →  <stmt>
         |  <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var>  → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var>
        | const
```

a = b + c ?

Copyright © 2018 Pearson. All rights reserved.

An Example Grammar for a small Language

<program> → <stmts>
<stmts> → <stmt>
| <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var>
| const

a = b + c ? <stmt> -> <var> = <expr>

<stmt>
→ <var> = <expr> (using <stmt> -> <var> = <expr>)
→ a = <expr> (using <var> -> a)
→ a = <term> + <term> (using <expr> -> <term> + <term>)
→ a = <var> + <term> (using <term> -> <var> for the first <term>)
→ a = b + <term> (using <var> -> b)
→ a = b + <var> (using <term> -> <var> for the second <term>)
→ a = b + c (using <var> -> c)

Copyright © 2018 Pearson. All rights reserved.

Derivation example

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

- Leftmost derivation of statement

  A = B * (A + C)

Copyright © 2018 Pearson. All rights reserved.

Derivation example

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

- Leftmost derivation of statement

  A = B * (A + C)

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

Copyright © 2018 Pearson. All rights reserved.

An Example Grammar for a small Language

```
<program> → <stmts>
<stmts> →  <stmt>
         |  <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var>  → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var>
        | const
```

a = b ?

a = b + c + d ?

Copyright © 2018 Pearson. All rights reserved.

Examples (in Racket)

- Non-terminals: <pointy brackets>.
- Terminals or literals: Concrete syntax, like {, }, and + below. They are so-called because they stand for themselves and can't be expanded further.

```
<expr> ::= <num>
         | {+ <expr> <expr>}
```

- 1
- {1}
- x
- {x}
- {+ x 1}
- {+ 1 2}
- {+ {+ 1 2} 3}
- {+ x x}

Copyright © 2018 Pearson. All rights reserved.

Extending the language

<expr> ::= <num>
| {+ <expr> <expr>}
| {let1 {<var> <expr>} <expr>}

let1 creates a new local binding of one variable, evaluates the body using that binding. Check if they are legal

- 1
- {1}
- x
- {x}
- {+ x 1}
- {+ 1 2}
- {+ {+ 1 2} 3}
- {+ x x}
- {let1 {x 1} 2}
- {let1 {x {+ 1 2}} {+ 3 4}}

Extending the language

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
```

- 1 yes <expr> ::= <num>
- {1} NO there is not {<num>}
- x: NO, we don't have <expr> ::= <var>
- {x} NO there is not {<num>}
- {+ x 1} NO Tries to match: {+ <expr> <expr>}, But x is not an <expr>
- {+ 1 2} YES Matches: {+ <expr> <expr>}, because there is not <expr> ::= <var>
- {+ {+ 1 2} 3} YES Matches {+ <expr> <expr>}
- {+ x x} NO, because, x is not an expression <expr>:=<num>
- {let1 {x 1} 2} YES, Match {let1 {<var> <expr>} <expr>}
- {let1 {x {+ 1 2}} {+ 3 4}}, YES {let1 {<var> <expr>} <expr>}

Examples

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
```

What can I see about these programs?

Program 1

```
{let1 {x 1}
 {+ x x}}
```

Program 2

```
{let1 {x 1}
    {let1 {y 2}
     {+ x y}}}
```

Copyright © 2018 Pearson. All rights reserved.

Examples

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         |  <var>
```

What can I see about these programs?

Program 1 yes

```
{let1 {x 1}
   {+ x x}}
```

Program 2 yes

```
  {let1 {x 1}
        {let1 {y 2}
          {+ x y}}}
```

Copyright © 2018 Pearson. All rights reserved.

Example

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         |  <var>
```

{let1 {x 1} {let1 {y 2} {+ x y} } }

{let1 {<var> <expr>} <expr>}
{let1 {x 1} <expr>}
{let1 {x 1} {let1 {y 2} <expr>}
{let1 {x 1} {let1 {y 2}{+ <expr> <expr>}
{let1 {x 1} {let1 {y 2}{+ x y}

Copyright © 2018 Pearson. All rights reserved.

Derivations

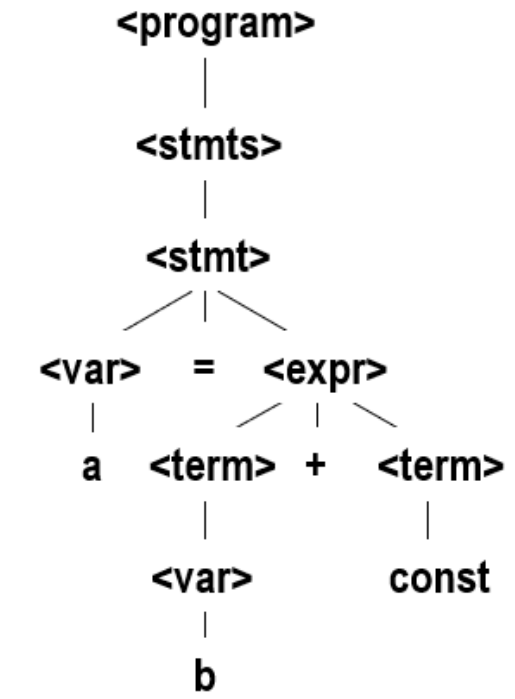
- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Copyright © 2018 Pearson. All rights reserved.

Parse tree representation

hierarchical representation of a derivation

```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```



Copyright © 2018 Pearson. All rights reserved.

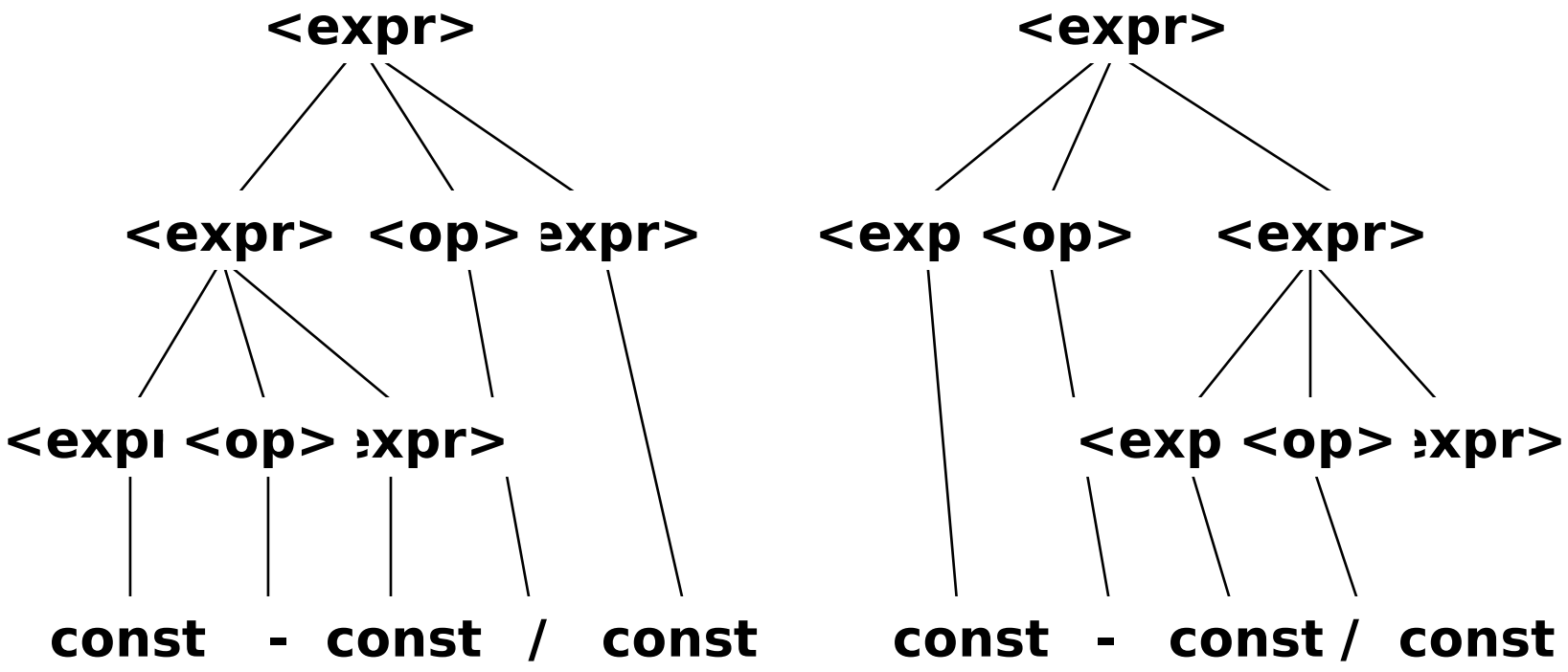
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

Copyright © 2018 Pearson. All rights reserved.

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const
<op> → / | -

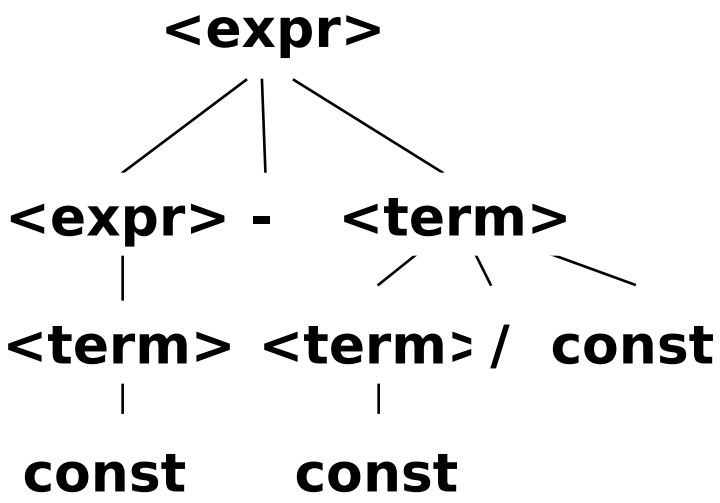


Copyright © 2018 Pearson. All rights reserved.

An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```

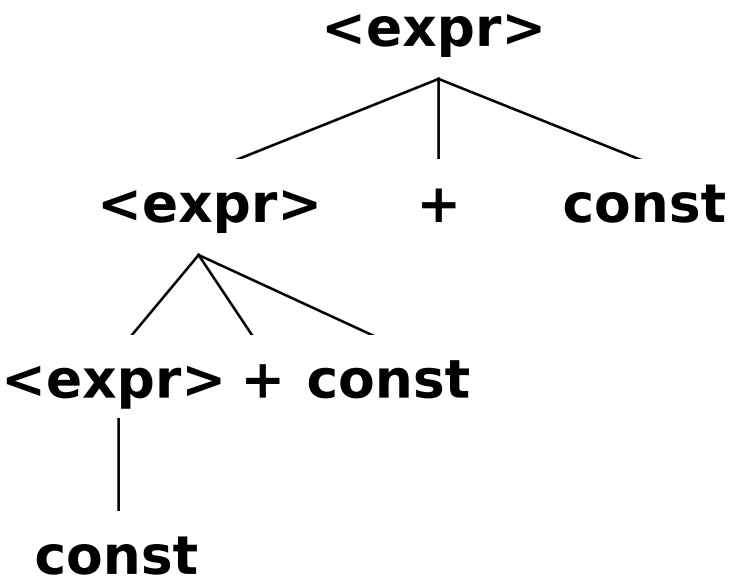


Copyright © 2018 Pearson. All rights reserved.

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Copyright © 2018 Pearson. All rights reserved.

Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> -> ident [(<expr_list>)]`
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`
- Repetitions (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

Copyright © 2018 Pearson. All rights reserved.

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
         | <expr> - <term>
         | <term>
 <term> → <term> * <factor>
         | <term> / <factor>
        | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
 <term> → <factor> {(* | /) <factor>}
```

Copyright © 2018 Pearson. All rights reserved.

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of =>
- Use of `opt` for optional parts
- Use of `oneof` for choices

Copyright © 2018 Pearson. All rights reserved.

Static Semantics

- Nothing to do with meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:
  - For each grammar symbol *x* there is a set *A(x)* of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $i <= j <= n$, define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars: An Example

- Syntax

  <assign> -> <var> = <expr>

  <expr> -> <var> + <var> | <var>

  <var> A | B | C

- actual_type: synthesized for <var> and <expr>
- expected_type: inherited for <expr>

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammar (continued)

- Syntax rule: <expr> → <var>[1] + <var>[2]
  Semantic rules:
  <expr>.actual_type ← <var>[1].actual_type
  Predicate:
  <var>[1].actual_type == <var>[2].actual_type
  <expr>.expected_type == <expr>.actual_type

- Syntax rule: <var> → id
  Semantic rule:
  <var>.actual_type ← lookup (<var>.string)

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Copyright © 2018 Pearson. All rights reserved.

Attribute Grammars (continued)

<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type

Copyright © 2018 Pearson. All rights reserved.

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Copyright © 2018 Pearson. All rights reserved.

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed

Copyright © 2018 Pearson. All rights reserved.

Operational Semantics

- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent

Copyright © 2018 Pearson. All rights reserved.

Operational Semantics (continued)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Copyright © 2018 Pearson. All rights reserved.

Operational Semantics (continued)

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages

- Two different levels of uses of operational semantics:
  - Natural operational semantics
  - Structural operational semantics

- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g.,VDL)

Copyright © 2018 Pearson. All rights reserved.

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Copyright © 2018 Pearson. All rights reserved.

Denotational Semantics - continued

- The process of building a denotational specification for a language:

  - Define a mathematical object for each language entity

  – Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

- The meaning of language constructs are defined by only the values of the program's variables

Copyright © 2018 Pearson. All rights reserved.

Denotational Semantics: program state

- The state of a program is the values of all its current variables

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

Copyright © 2018 Pearson. All rights reserved.

Decimal Numbers

```
<dec_num> →  '0' | '1' | '2' | '3' | '4' | '5' |
             '6' | '7' | '8' | '9' |
             <dec_num> ('0' | '1' | '2' | '3' |
                        '4' | '5' | '6' | '7' |
                        '8' | '9')
```

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Copyright © 2018 Pearson. All rights reserved.

Expressions

- Map expressions onto Z ∪ {error}
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

Copyright © 2018 Pearson. All rights reserved.

Expressions

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                                 undef)
              then error
           else
           if (<binary_expr>.<operator> == '+' then
              Me(<binary_expr>.<left_expr>, s) +
                     Me(<binary_expr>.<right_expr>, s)
           else Me(<binary_expr>.<left_expr>, s) *
               Me(<binary_expr>.<right_expr>, s)
        ...
```

Copyright © 2018 Pearson. All rights reserved.

Assignment Statements

- Maps state sets to state sets U {error}

M_a(x := E, s) Δ=
    if M_e(E, s) == error
       then error
       else s' =
  {<i_1,v_1'>,<i_2,v_2'>,...,<i_n,v_n'>},
                where for j = 1, 2, ..., n,
                    if i_j == x
                      then v_j' = M_e(E, s)
                      else v_j' = VARMAP(i_j, s)

Copyright © 2018 Pearson. All rights reserved.

Logical Pretest Loops

- Maps state sets to state sets U {error}

```
Ml(while B do L, s) Δ=
    if Mb(B, s) == undef
        then error
        else if Mb(B, s) == false
            then s
            else if Msl(L, s) == error
                  then error
                  else Ml(while B do L, Msl(L, s))
```

Copyright © 2018 Pearson. All rights reserved.

Loop Meaning

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

  - Recursion, when compared to iteration, is easier

    to describe with mathematical rigor

Copyright © 2018 Pearson. All rights reserved.

Evaluation of Denotational Semantics

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it are of little use to language users

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called *assertions*

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics (continued)

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- An example
  - `a = b + 1  {a > 1}`
  - One possible precondition: `{b > 10}`
  - Weakest precondition: `{b > 0}`

Copyright © 2018 Pearson. All rights reserved.

Program Proof Process

- The postcondition for the entire program is the desired result
  - Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics: Assignment

- An axiom for assignment statements (x = E): $\{Q_{x->E}\}$ x = E {Q}

- The Rule of Consequence:

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics: Sequences

- An inference rule for sequences of the form S1; S2

  {P1} S1 {P2}
  {P2} S2 {P3}

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics: Selection

- An inference rules for selection
  - **if** B **then** S1 **else** S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics: Loops

- An inference rule for logical pretest loops

  {P} **while** B **do** S **end** {Q}

  where I is the loop invariant (the inductive hypothesis)

Copyright © 2018 Pearson. All rights reserved.

Axiomatic Semantics: Axioms

- Characteristics of the loop invariant: I must meet the following conditions:
  - P => I -- the loop invariant must be true initially
  - {I} B {I} -- evaluation of the Boolean must not change the validity of I
  - {I and B} S {I} -- I is not changed by executing the body of the loop
  - (I and (not B)) => Q -- if I is true and B is false, Q is implied
  - The loop terminates -- can be difficult to prove

Copyright © 2018 Pearson. All rights reserved.

Loop Invariant

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Copyright © 2018 Pearson. All rights reserved.

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Copyright © 2018 Pearson. All rights reserved.

Denotation Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Copyright © 2018 Pearson. All rights reserved.

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational

Copyright © 2018 Pearson. All rights reserved.